A Super-Scheduler for Embedded Reconfigurable Systems

S. Ogrenci Memik **E. Bozorgzadeh** **R. Kastner** **M. Sarrafzadeh**
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095
{seda, elib, kastner, majid}@cs.ucla.edu

Abstract— Emerging reconfigurable systems attain high peformance with embedded optimized cores. For mapping designs on such special architectures, synthesis tools, that are aware of the special capabilities of the underlying architecture are necessary. In this paper we are proposing an algorithm to perform simultaneous scheduling and binding, targeting embedded reconfigurable systems. Our algorithm differs from traditional scheduling methods in its capability of efficiently utilizing embedded blocks within the reconfigurable system. Our algorithm can be used to implement several other scheduling techniques, such as ASAP, ALAP, and list scheduling. Hence we refer to it as a super-scheduler. Our algorithm is a path-based scheduling algorithm. At each step, an individual path from the input DFG is scheduled. Our experiments with several DFG's extracted from MediaBench suit indicate promising results. Our scheduler presents capability to perform the trade-off between maximally utilizing the high-performance embedded blocks and exploiting parallelism in the schedule.

1. INTRODUCTION

In the past decade substantial evidence has been provided by the research community as well as commercial products regarding the advantages of configurable and reconfigurable systems [1, 2, 3]. Currently, reconfigurable fabric is considered to be part of hybrid systems such as System On Chip (SoC) solutions. One trend is towards integration of highly optimized hard cores and hardwired blocks with reconfigurable fabric. The main goal here is to utilize the optimized blocks to improve the system performance. Such programmable devices are targeted for a class of applications, such as DSP [2], networking or data communications [4]. The flexible programmable logic can be supported with the high-density high-performance cores at various levels, such as functional block level [4] or at the level of basic arithmetic operations, e.g., multipliers [2]. The power of the context-based reconfigurable architectures lies in the efficient utilization of the fixed cores within the system. In this paper we propose a simultaneous scheduling and binding algorithm for context-specific systems containing multiple fixed cores. Our scheduler uses a path-based algorithm. Scheduling on each path is done by solving optimally the *max-weighted-chain problem* [5]. This algorithm constitutes a *super-scheduler*, in the sense that with proper parameterization it can be reduced to other fundamental scheduling methods, such as ASAP, ALAP, list scheduling, etc. Moreover, our scheme enables incremental generation of a schedule from a given partial schedule. Our scheduler is aware of the customization of the target architecture. The scheduler performs a trade-off between reuse of the embedded cores and achievable parallelism on hardware. The time complexity of the proposed scheduling algorithm is $O(N^2)$, where N is the number of operations in the input DFG.

The rest of the paper is organized as follows. Section 2 states the scheduling problem and presents our algorithm. In Section 3 results are presented. We discuss our conclusions and future work in Section 4.

2. SCHEDULING

Given a reconfigurable architecture with several embedded cores, as described in the introduction, a high-level synthesis tool maps the design onto it. In the following sections we briefly describe the overall flow and present our solution to the scheduling problem within the high-level synthesis framework. In Section 2.1 the scheduling problem for embedded reconfigurable systems is introduced. In this section we also discuss related work. The definition of the problem and our proposed algorithm are presented in Section 2.2.

2.1 Overview and Related Work

In this work we present a scheduling algorithm for reconfigurable architectures containing fixed blocks. The operations in a given application are scheduled such that the available embedded cores are utilized in the best way to improve performance. The architecture would be customized with these cores for a particular family of applications. Hence, when implementing one of those applications the critical operations common to this family are most likely to be executable by the embedded cores. Within one application the number of such operations are expected to be high as well. Our scheduler generates schedules while maintaining a balance between exploiting the optimized embedded cores and avoiding the limited availability of these blocks become a bottleneck for the end result. Functional units for any desired operation type can be instantiated using reconfigurable logic. For operations that cannot be performed by the embedded cores this is a necessity. For other operations this can be done in order to exploit parallelism in the schedule. However, as mentioned earlier the available blocks are highly preferred for those operations.

Our method uses the idea of producing schedules for individual paths within a DFG. A similar scheme is used by other path-based scheduling techniques [6, 7]. In [6] each possible path is scheduled independently in an optimal fashion. Then the schedules for each path are overlapped, again in an optimal way. To allow the optimal scheduling of all execution paths, operations may be scheduled into several states. In our approach, one path of the input is scheduled at a time and the schedule of each path is constrained by the partial schedule constructed so far. This scheme provides the synthesis tool the ability to do incremental synthesis and incremental change. The result may not be guaranteed to be optimal in all cases, but compared to other path-based scheduling algorithms [6, 7], our algorithm is more time-efficient. The algorithm in [6] uses a method for optimally solving a sub-problem that has exponential complexity in the worst case. Subproblems in our algorithm are solved optimally in polynomial time. Also both in [6] and [7] an exponential number of paths are considered. Whereas in our case, each path's schedule is fixed after one attempt and the algorithm proceeds until all operations are scheduled. Another algorithm proposed in [8] is

based on bipartite graph matching. In this work a bipartite matching is performed and the result is pruned with a heuristic in order to comply with precedence constraints. Our approach to the problem is different, such that we generate bipartite graphs for matching by observing dependencies to begin with. Then each matching problem is solved optimally in polynomial time. In this paper we show how to optimally solve the maximum-weight-matching problem while satisfying all precedence constraints in polynomial time.

Our algorithm can be easily reduced to basic scheduling algorithms, such as ASAP, ALAP, and list scheduling by proper parameterization. We will elaborate on this in Section 2.3.

2.2 Our Scheduling Algorithm

Given a Data Flow Graph (DFG) as input, the problem of simultaneous scheduling and binding is assigning a clock cycle as a start time and a resource to each operation in the DFG. A schedule is valid if:

- For each operation a valid start time and a binding is defined.
- Data dependencies imposed by the DFG are not violated: $\forall(op1, op2), start_{op_2} \geq finish_{op_1}$
- Each resource can perform *at most* one operation at any given clock cycle: $\forall op_i, \forall op_j, if start_{op_i} \leq start_{op_j} < finish_{op_i} then$ $Binding(op_i) \neq Binding(op_j)$.

Our method provides an incremental scheduling scheme by handling one path within the given DFG at a time. The global problem is first divided into sub-problems of scheduling individual paths. The sub-problem is then solved with a bipartite graph matching approach.

Selection of Paths in the DFG: Paths are selected from the input DFG according to criticality. After a selected path is scheduled, the schedules of the operations on it are fixed. A new path is selected in the next iteration. The schedule of each newly selected path depends on the existing partial schedule.

Scheduling of Paths using Non-Crossing Bipartite Matching: Given a DFG and a target architecture the first step is to setup a *resource assignment table*. The rows in this table correspond to clock cycles and each column corresponds to a hardware resource. The architecture specifications provide information on the types and numbers of fixed blocks. For each of those a column is generated. The table is extended with new columns as reconfigurable modules are added to the binding set. If an operation is scheduled at a certain cycle on a hardware resource, the corresponding entry in the table is marked as well as consecutive cycles within the same column for the duration of the operation's execution time.

The problem instance for a path P is then converted to a bipartite graph representation $B(V, E)$, where
$V = O \cup C,\ O = \{op_i \mid op_i \in P\}\ and\ C = \{cycle_i \mid 1 \leq i \leq max_{cycles}\}$
$E = \{(u, v) \mid u \in O, v \in C\}$
An example of a possible representation is depicted in Figure 1(a). Every edge in B is associated with a weight w. Edges of B are generated while obeying the data dependencies between operations on the same path. In addition, dependencies among already scheduled operation in previous path and the currently scheduled path are considered. The last issue to consider is related to the availability of hardware resources. If the algorithm is applied to a model with fixed amount of resources, then the following holds:
If for an operation u there are no available resources at cycle i, then there cannot be an edge between u and i. This information is obtained from the resource assignment table.

Each edge is assigned a weight w that represents conceptually the following:

- From a single operation's perspective, the weight w of an edge $e = (u, v)$ represents the tendency of operation u to be scheduled at clock cycle v. Each operation would preferably be scheduled at the clock cycle onto which an edge from the operation is incident with highest weight.
- Comparing several operations with connecting edges to the same clock cycle, the operation with highest edge weight would have highest priority.

We generate weights for the edges in the bipartite graph considering several factors. The most obvious one being the tendency of operations to be scheduled *early*. Since the primary goal is to obtain a schedule with low latency the algorithm should not delay the schedule of operations. In addition, since fixed cores are optimized blocks, matching operations are desired to be executed on those. When considering two different cycles for possible start time of an operation, the one with an available fixed block would be preferred over another at which no optimized fixed block is free. In this manner the customized features of the target architecture can be exploited best. Similarly further considerations can be incorporated into the computation of edge weights. A combination of various cost metrics can be tuned according to the current needs.

The solution to the scheduling problem of a given path P is now finding a non-crossing matching on the constructed bipartite graph such that the sum of edge weights are maximized. We find this matching by converting the problem into a geometric representation. For each possible matching indicated by an edge in the bipartite graph, a point in the x-y plane is created. For $e = (u, v)$ a point $p(x, y)$ is created, such that $x = index(u)$ and $y = index(v)$. This is shown in Figure 1(b). The weight of each edge is transferred as a weight to the corresponding point in the plane. The problem of finding a maximum-weight non-crossing matching is now equivalent to finding a maximum-weight chain of length k, where k is equal to the number of operations on path P. Within this chain each point *dominates* the preceding point, hence dependencies are observed.

Definition 1. *Point* $p = (p_x, p_y)$ *dominates point* $q = (q_x, q_y)$ *iff* $p_x > q_x \ \wedge p_y > q_y$.

Definition 2. *Each point p in the plane is associated with a level, such that none of the points in the same level dominate each other. Points in higher levels dominate points in lower levels.*

Definition 3. *The level associated with a point p in the plane is equal to the highest level among the points dominated by p incremented by one. The origin is assumed to have level* 0.

Lemma 2.1. *There are exactly k levels in the resulting point set in maximum-weight non-crossing matching problem and two points have same level iff they have the same x-coordinate.*

Proof: Since there are only k possible different x-coordinates, there can only be k possible levels. Every x-coordinate corresponds to one of the k operations on path P. The points with same x-coordinate, but different y-coordinates have the same level, since none of them can dominate another. Let two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ be at different x-coordinates, such that $x_1 < x_2$, i.e. on the path P, operation corresponding to p_1 precedes operation corresponding to p_2. In order for these points to have the same level y_2 must be less then or equal to y_1, otherwise p_2 would have dominated p_1. There can not exist such a point p_2, because the first rule of edge generation prohibits assignment of an edge to p_2 anytime equal or earlier to the

first cycle that p_1 was matched with an edge. That means any point at x-coordinate x_2 having smallest y-coordinate would dominate at least one point at x-coordinate x_1.□

According to Lemma 3.1, a level value to each point in the plane can easily be assigned, where the level of each point is equal to its x-coordinate. A polynomial time algorithm for finding the maximum-weight chain is proposed in [5]. In the case of assigning unit weights to each point this algorithm returns the longest chain, which naturally corresponds to the maximum sum of weights. However, when arbitrary weights are assigned to the points in the plane, this algorithm does not necessarily yield a chain of length k. Another procedure proposed by [7] *only* creates a chain of length equal to the number of operations by using the level information. The *max_weighted_k_chain()* procedure is as follows:

max_weighted_k_chain()

```
1 initialize labels of points at level = 1 to their weights and
labels of points at other levels to 0
2 for level = 2 to k do
3     for each point in level i do
4         max_label := maximum label found at previous level
          among dominated points
5         label(point) := label(point)+max_label
6         assign pointer from point to the dominated point
          providing the max_label found to point
7     end for
8 end for
```

The max-weighted chain construction starts with the point p_k at level k with highest label. p_k is added to the max_chain as the p_kth element. The pointer created after the update of p_k's label links it to a point p_{k-1}. By tracing this pointer, p_{k-1} is located and added to the max_chain as the $k-1$th element. This is continued until the last pointer points to the first element of the max_chain which is at level = 1.

Theorem 2.1. *max_weighted_k_chain() produces the optimal k-chain.*

Proof: When computing the labels of the points at $level = 2$ each point is linked to a unique point with largest label at $level = 1$, such that the sum of the weights of the two points is maximum and the dominance relation holds. The maximum label among all points at level 2 would indicate the maximum possible sum of weights for a 2-point chain. When the algorithm proceeds to the next level, labels at the new level will be computed using the partial sums computed in the earlier levels. We know that those partial sums were the maximum possible values while maintaining dominance condition in the chain. Since each point at the new level will pick the maximum partial sum carried from the immediate lower level, the new partial chain sums (labels) will remain maximal. By induction on the number of levels, at kth level, the point with maximum label indicates the maximum sum of weights of a k-chain.□

2.3 Super-Scheduler Property

Our scheduling algorithm is reducible to other existing scheduling algorithms. The selection of (*partial*) paths and assignment of weights to the edges in the bipartite graph are the two key issues here. We will demonstrate this conversion property on two scheduling techniques:

As Soon As Possible (ASAP) Scheduling: To the edges incident on one operation Op_i, a monotonically decreasing series of weights is assigned. The edge connecting Op_i to the earliest clock cycle receives the highest weight and from there on each edge weight to the next cycle is ϵ less then the previous one. Here edge weights

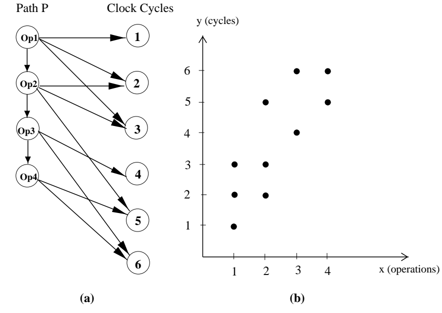

Figure 1: **Bipartite Graph Representation**

only reflect the desire of the operations to be scheduled *early*. Each operation can be assigned to the earliest possible cycle designated with the highest weight in the matching.

List Scheduling: In list scheduling, operations in the input DFG are ordered according to some priority function The algorithm assigns operations to clock cycles by tracing this ordered *list* and obeying dependencies and resource constraints. In a unit delay case each operation's level would be the length of the longest path to the output. In another case, operations may be ordered according to their slack (ALAP-ASAP). In this setup our incremental path-based scheme can be utilized with the help of a minor addition. We add a data structure that contains information on the number of operations at each priority level, separate numbers for each operation type per level. The algorithm selects the most critical path. Each operation on the path is checked as to whether it can be scheduled at this step. This depends on the following; the operation does not have any predecessor that is not in the path or already scheduled, no operation at a higher priority level exists or no operation of the same type at higher priority level exists. Starting from the first operation in the path, the longest path of operations with qualifying properties are selected and matching-based scheduling is applied on this path.

2.4 Complexity of Our Algorithm

The core of our algorithm is the maximum-weight k-chain procedure. In this procedure, at each level, the number of points are bounded by $O(C_{max})$, where C_{max} is the maximum number of clock cycles included in the bipartite graph. The number of levels is equal to the number of operations on the given path P. If we denote the number of operations on P with p, then $n = O(pC_{max})$. The complexity of *max_weighted_k_chain()* becomes $O(pC_{max})$. This procedure is repeated until all operations in the input DFG are scheduled.

Theorem 2.2. *The complexity of geom_Scheduling (DFG, IP_Library, reconfigurableResource, costFunction(Parameter_List)) is* $O(N^2)$, *where N denotes the number of operations in the DFG.*

Proof: If at every step i a path containing p_i operations are scheduled, then the total time to schedule a DFG can be expressed as

$$\sum_{i=1}^{i=max_{step}} p_i \cdot C_{max},$$

which is equal to

$$C_{max} \cdot \sum_{i=1}^{i=max_{step}} p_i \; and \; \sum_{i=1}^{i=max_{step}} p_i = N.$$

Assuming each operation in the DFG can be performed within a constant number of clock cycles, the maximum number of clock

cycles required to schedule a DFG is bounded by the number of operations with a constant coefficient k. $C_{max} = O(kN)$. Eliminating the constant k, the complexity of the algorithm becomes $O(N \cdot N) = O(N^2)$. □

3. RESULTS

In this section we will present our experimental results. We have used applications from the MediaBench suite [10] to test our algorithm. We have selected function blocks from applications contained in the suit and generated DFGs for them using the SUIF2 compiler system [9]. Table 1 shows the files, from which input DFGs were generated. The corresponding applications containing these files are given as well.

Individual DFGs are scheduled with two different methods shown in Table 2. For each application a set of hardware resources are specified. Within the available set of resources there are multiple components with same functionality, but different delays. This represents the existence of optimized cores for certain operations. The slower versions of such resources, in turn, correspond to instances created on reconfigurable logic. Several DFGs result from each functional block. We have enumerated those, and labeled the DFGs with a unique index, such as DFG20. In Table 2 scheduling results for a number of selected DFGs are given. The rest were mostly very simple DFGs in terms of number of nodes and depth. We compare the results of our algorithm against the results obtained from the linear programming solver, CPLEX. The scheduling problem has been described as a linear integer program for our problem instance. The objective function of the integer linear model tries to minimize the latency and the number of slow blocks used. A higher priority is given to latency minimization. Similarly in our algorithm, we try to minimize the latency, while trying to utilize the optimized blocks as well as possible. CPLEX provides us an optimal solution for the given objective function. Therefore we are comparing our results to those generated by the linear solver. The latency columns report the latencies of the scheduled DFGs, while the second column and the fourth column represent the core usage for CPLEX and our algorithm respectively. For each DFG the total number of operations assigned to optimized blocks is given vs. the total number of operations that could have been performed by any available optimized block.

As depicted in Table 2 our algorithm was able to produce latencies compatible with CPLEX results for most of the cases. In only three DFGs our algorithm could not achieve optimality. However, in two of these cases, convolve-DFG98 and fft-DFG18, our algorithm was able to utilize the optimized blocks as good as CPLEX results. In the case of fft-DFG27 the increase in latency resulted from a trade-off aiming to increase usage of optimized blocks. For this DFG, our algorithm was able to outperform the optimized block utilization obtained from CPLEX solution.

Benchmark	C File	Description
epic	convolve.c	2D image convolution
rasta	fft.c	Fast Fourier Transform
mpeg2	getblk.c	DCT Block Decoding
mpeg2	motion.c	Motion Vector Decoding

Table 1: Origins of DFGs from MediaBench

Benchmark	CPLEX		Our Algorithm	
convolve	latency	core usage	latency	core usage
DFG0	18	5/6	18	5/6
DFG19	12	2/3	12	2/3
DFG98	18	5/6	20	5/6
fft				
DFG18	26	6/9	28	6/9
DFG27	22	5/6	27	6/6
getblk				
DFG42	14	3/4	14	3/4
DFG91	18	4/4	18	3/4
motion				
DFG6	-	-	41	22/26
DFG7	24	6/8	24	6/8

Table 2: Scheduling results in terms of DFG latencies, in cycles. "-" indicates result could not be obtained in a reasonable amount of time Usage of cores indicates the total number of operations actually assigned to optimized blocks vs. the total number of operations that could have been performed by any available optimized block

4. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a scheduler to perform simultaneous scheduling and binding. Our algorithm is mainly targeted for reconfigurable systems, where multiple optimized cores are available within the system. We have shown that sub-steps of our algorithm can be solved optimally in polynomial time. Also our proposed method forms a framework as a super-scheduler. Within this framework several scheduling techniques, such as ASAP, ALAP, and list scheduling can be implemented. Our preliminary experiments indicate good results in terms of the combined effort of minimizing latency and utilizing optimized cores available in a given system.

We are currently integrating our algorithm into a high-level synthesis tool. After complete automation of the algorithm we aim to experiment with larger benchmarks from other applications. Benchmarks available at this time were not very large. Therefore, investigation of appropriate benchmarks both in context and in size is an important task to accomplish.

5. REFERENCES

[1] S. Hauck, "The Role of FPGAs in Programmable Systems ," *Proceedings of the IEEE* , Vol. 86, No. 4, pp. 615-638, April, 1998.

[2] Xilinx Inc., http://www.xilinx.com.

[3] Altera Inc., http://www.altera.com/products/devices/excalibur/exc-index.html.

[4] Lucent Technologies, "Lucent Technologies Announces High-Speed Communications Cores For Customizing ORCA(r) FPGAs", http://www.lucent.com/micro/NEWS/PRESS97/081897a.html.

[5] M.J. Atallah, S.R. Kosaraju, "An Efficient Algorithm for Maxdominance with Applications," *Algorithmica* , 4(1989), 221-236. Feb 2000.

[6] R. Camposano, "Path-Based Scheduling for Synthesis," *IEEE Transactions on Computer-Aided Design*, 10(1), January 1991.

[7] S. Raje, M. Sarrafzadeh, "GEM: A Geometric Algorithm for Scheduling ," *1993 International Symposium on Circuits and Systems (ISCAS-93)* , May 1993.

[8] A. H. Timmer, J. A. G. Jess, "Exact Scheduling Strategies based on Bipartite Graph Matching," *Proceedings of the European Design and Test Conference*, March, 1995.

[9] Stanford University Compiler Group, "The SUIF 2 Compiler System," *http://suif.stanford.edu/suif/suif2/index.html*.

[10] C. Lee, M. Potkonjak, W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *Proc. of International Symposium on Microarchitecture, IEEE Micro-30*, 1997.